Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation

USING META-LEVEL DESCRIPTIONS FOR SELECTIVE APPLICATION OF MULTIPLE REWRITE RULES IN ALGEBRAIC MANIPULATION

by

Alan Bundy & Bob Welham

DAI RESEARCH PAPER NO. 121

Paper submitted to: *Artificial Intelligence Journal*, 1979

Also produced as *DAI Working Paper No. 55*, 1979

USING META-LEVEL DESCRIPTIONS FOR SELECTIVE APPLICATION OF MULTIPLE REWRITE RULES IN ALGEBRAIC MANIPULATION

by

Alan Bundy & Bob Welham

Abstract

This paper describes the PRESS algebra system, a computer program which solves equations and inequalities and simplifies expressions. We argue that multiple rewrite rules, selectively applied, using meta-level reasoning and descriptions to guide that application are a powerful tool in automatic theorem proving. Our reasons are that compared to conventional, homogeneous, exhaustively applied, rewrite rule sets, selectively applied, multiple rule sets are easier to design and analyse, lead to less search and lend themselves to automatic learning.

Acknowledgements

Our thanks are due to Ira Goldstein and Soei Tien Tan for some of the original inspiration; Gordon Plotkin, Joel Moses, Alan Robinson, Woody Bledsoe and John Seeley Brown for encouragement and to many other people whose ideas we have unwittingly absorbed over the years.

This work was supported by SRC grants B/SR/22993 and B/RG/94493 and an SRC studentship to Bob Welham.

Keywords

rewrite rules, theorem proving, mathematical reasoning, algebraic manipulation and meta-level reasoning.

Table of Contents

1. Introduction

This report describes PRESS (PRolog Equation Solving System), a computer program for solving equations and doing other kinds of algebraic manipulation, on expressions involving: polynomial; trigonometric; exponential and logarithmic functions.[1] It is based on ideas originally expounded in [Bundy 79] and [Bundy 75].

The program was largely written during 1975 by Bob Welham in the language Prolog (see [Pereira et al 78]). It consists of approximately 250 clauses and occupies 13K, 36 bit, Dec10 words. The Prolog system itself occupies a further 20k words.

The main use of PRESS has been as the algebraic package of the MECHO program (see [Bundy et al 79]). It has been extended from its original purely equation solving role to handle problems which have arisen in the MECHO project, namely inequality handling and the use of semantic information. Suprisingly, some of the techniques originally developed for equation solving have also found application in inequality handling.

But PRESS was not developed as a service program for MECHO. It was intended as a vehicle to explore some ideas about controlling search in mathematical reasoning using meta-level descriptions and strategies. The lessons learnt in the development of PRESS can be summed up as follows:

- Rewrite rule systems have again been shown to be a powerful technique[2] in Mathematical Reasoning.

- More power can be gained by the selective use of several different rewrite rule systems, each designed to do a different job.

- Meta-level/syntactic descriptions can be used to good effect to both design the rewrite rule systems and to decide automatically which rule system to apply and how to apply it.

---

[1] The expressions manipulated are called the R Elementary Expressions. A BNF definition of them is given in appendix I.

[2] By powerful we mean that the ratio of theorems proved to total search involved, is high.

2. Overview

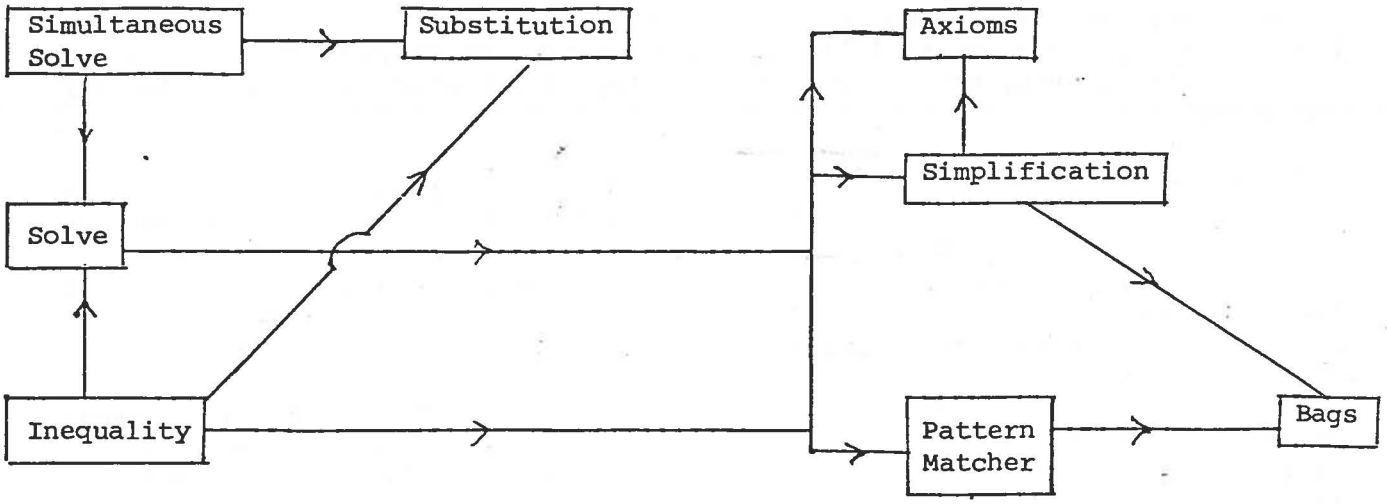

Figure 2-1: Block Diagram: The main modules of PRESS

In this report we will regard the PRESS program as being divided into a number of modules (represented as boxes in figure 2-1 above). These modules are major groupings of Prolog clauses which communicate with each other via pattern directed invocation. The arrows between the boxes represent subroutine calls. The role of each module is explained below.

2.1 Simultaneous Solve

The main procedure is **simsolve** which takes a conjunction of equations and a set of unknowns and solves the equations in terms of the unknowns. The method called 'stripping' in [Bundy 79] is used. An equation and an unknown, x say, are chosen and the equation is solved in terms of x (treating other unknowns as constants) using the Solve module. This solution is then substituted into the remaining equations using the Subst module. The process is called recursively on the remaining equations. The solutions obtained from this recursive call are then backsubstituted into the solution for x, using **subst**.

The **simsolve** procedure is only used by the MECHO program to solve conjunctions of equations generated by the Marples' algorithm. This algorithm is guaranteed to provide as many equations as unknowns and to produce them in such an order that a left to right pairing of equations and unknowns will be optimal for solution. Hence no time is currently spent checking or reordering them. These steps were provided in early versions of the program but have been deleted from the current version for the sake of efficiency.

2.2 Solve

The main procedure of the solve module, solve11, takes an equation and an unknown and produces a solution. To do this it uses six equation solving methods: Isolation; Collection; Attraction; Linear; Quadratic and change of unknown. These methods are the heart of the PRESS system and will be explained more fully in section 3.

The equation is analysed by a series of syntactic tests and this analysis is used to pick an appropriate method.

The methods currently provided constitute a basic equation solving ability, sufficient, for instance, for the MECHO program. It is planned to add further methods in the future to enable PRESS to handle more difficult problems. Outlines of these new methods can be found in [Bundy 75, Bundy 79].

2.3 Inequality

The job of the 'Inequality' module is to 'solve' conjunctions of inequalities for an unknown, x say, by isolating this unknown on one side of an inequality, e.g. from 2*g*r >= v^2 & g*r > v^2 produce r > v^2/g. This process is required for the MECHO program, especially for simplifying the inequalities produced by the motion predictor when solving 'roller coaster' problems.

Six methods are used:

- four based on four of the equation solving methods, namely, Isolation, Collection, Attraction and Change of Unknown, for 'solving' single inequalities for single unknowns.

- one for eliminating superflous unknowns from these 'solutions', using differentiation and stationary values.

- and one for reducing the length of conjunctions of inequalities.

All these methods are explained more fully in section 4.

2.4 Simplification

The simpliication procedures take an algebraic expression and return an equivalent or equal, simplified expression. There are four different simplify procedures:

- one, Normalize, for putting expressions into a weak normal form by replacing some functions by their definitions in terms of others;

- one, Tidy, for cleaning up the results of the solve and inequality modules, by replacing clumbersome expressions, e.g. x*0;

- one, Simplify, for proving that terms lie in certain ranges of the real line, e.g. between 0 and 90;

- and one, Prove, for proving simple theorems.

They work, mainly, by applying a system of rewrite rules to the expression to be simplified, but they also use an arithmetic evaluator, some special techniques for group operators (+, *, &, v) based on bags and semantic information about the quantities involved.

The simplification module is explained more fully in section 5.

2.5 Substitution

The procedure subst takes an expression and a substitution, applies the substitution to the expression and returns the resulting expression. To simplify some of the processes which use substitution (like simsolve above), substitutions are represented by equation solutions. That is the class of substitutions are defined as follows:

- x = t is a substitution, where x is an unknown and t is a term not containing x.
- If S1 and S2 are substitutions then S1 & S2 and S1 v S2 are substitutions.

To apply a disjunctive substitution each disjunct is applied separately and the two results are disjuncted together. To apply a conjunctive substitution the two conjuncts are applied in sequence. To apply a simple substitution, x = t, each occurence of x is replaced with t.

2.6 Axioms

Each of the modules: Solve, Inequality and Simplification, use sets of rewrite rules to manipulate algebraic expressions. These rewrite rules are stored as axioms and accessed via the pattern matcher described in section 2.7.

The axioms are all either equations (identities), equivalences or implications, sometimes with attached conditions. Each one is represented as a unit clause (examples can be found in appendix II). The predicate of this clause specifies which rewrite system the axiom is useful to - if an axiom is useful to more than one system it is represented more than once. The arguments vary from predicate to predicate, but represent such information as:

- The two expressions on either side of the equation, equivalence etc.
- The condition, if any.
- The variable which this axiom helps to Isolate, Collect etc.

Variables are represented by Prolog variables and constants by Prolog constants. Conditions are usually checked by a call to the Simplify method (see section 5).

2.7 Pattern Matcher

The match procedure takes two expressions and succeeds if they match. It is an extension of the Prolog unification algorithm because the associativity and commutativity of + and * are built-in. On the other hand the matcher is only one way, since it is in the nature of equation solving that no variables appear in the expression being manipulated but only in the axioms being applied (see [Bundy 79]).

Because the matcher need only be one way, the building-in of associativity and commutativity is particularly simple. We use the naive method of expressing sum and product terms as functions of bags (see section 2.8) and attempting to match the axioms against them in all possible ways.

For instance, suppose that the term

(a+x)+(y+b)

from an equation, were to be matched against the term

U + V

from an axiom[1]. The equation term would be re-expressed as a function of a bag, i.e.

[+, a, x, y, b]

and all the various splits of this bag, [+, a, y], [+, x, b] etc would be matched against U, V.

2.8 Bags

Two procedures are provided: **decomp** and **recomp**, which turn regular sums, products, conjunctions and disjunctions into functions of bags and back again, respectively. Lists are used to represent functions of bags, e.g. [+, a, x, b], [*, x, sin(x]). Other functions are also converted into the same format, e.g. [sin, theta].

The conversion is only done at the top level. That is, the top-most function is converted, unless it is a sum, product, conjunction or disjunction in which case subfunctions of the same type are also converted. For instance, sin(x+y) becomes [sin, x+y], but a+(x+y) becomes [+, a, x, y] and a+(x+(y*z)) becomes [+, a, x, y*z].

Functions of bags are re-converted in left-bracketed form, e.g. [+, a, x, y] is re-converted to (a+x)+y.

---

[1] Note that we use the Prolog convention that a word is a variable or a constant according as its first letter is in upper or lower case

3. The Solve Module

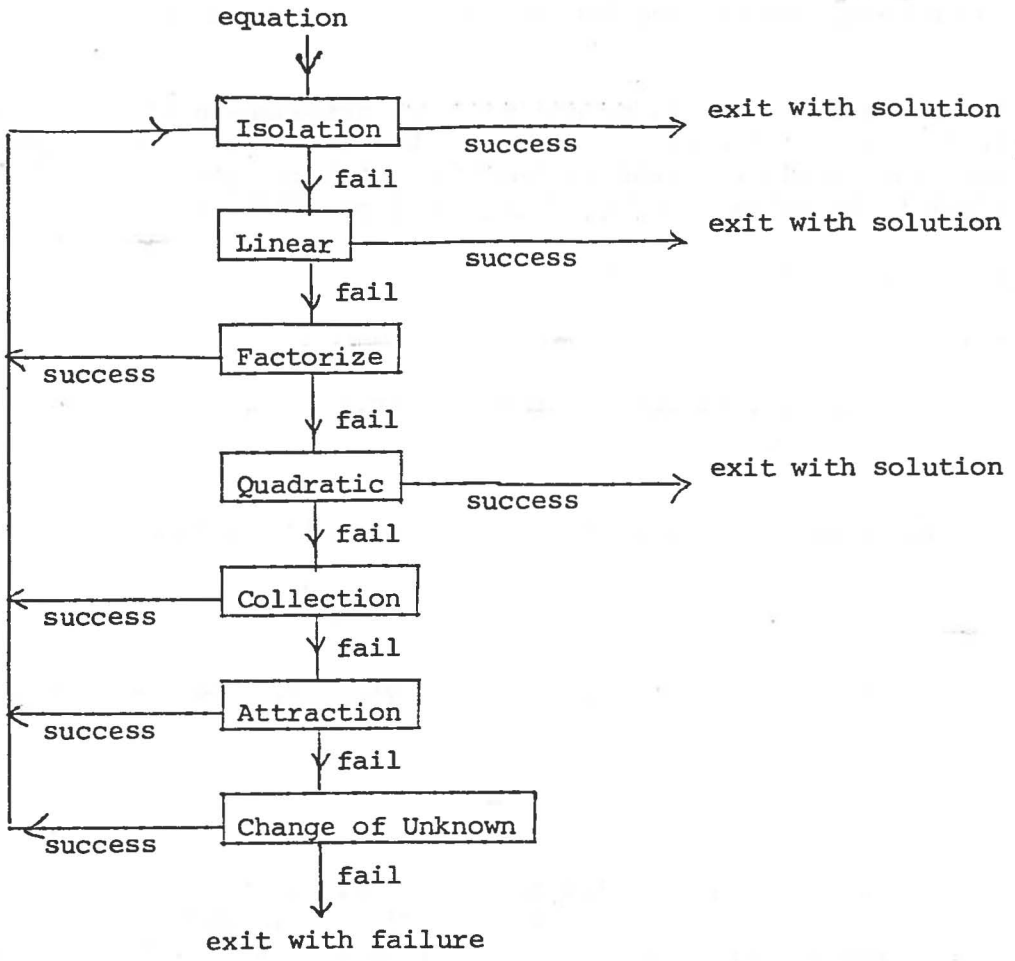

Figure 3-1: Flow Diagram: Methods of solving equations

In this section we describe the various methods used to solve one equation in one unknown. These methods are organised on the principles that methods requiring the least work are tried first.

The methods requiring least work are Isolation and Linear, so these are tried first. If these fail then an attempt is made to factorize the equation into two equations and each of these is solved recursively. The two answers obtained are disjuncted to form the answer to the original equation, trivial disjunctions being tidied to single equations.

Special cases are now attempted - the only one currently implemented being Quadratic, which, as its name implies, is for solving quadratic equations.

The rest of the Basic Method (see [Bundy 75]) is then tried. Collection

attempts to reduce the number of occurences of unknowns and Attraction to bring them closer together.

Finally, a very simple Change of Unknown strategy is used to try to simplify a complex equation to two simple ones.

3.1 Isolation

Isolation is only attempted on equations containing a single occurence of the unknown, x, e.g. on

```
log(e, x^2 - 1) = 3
```

but not on

```
exp(e,sin(x)) + exp(e,cos(x)) = a
```

On such equations it is guaranteed to succeed.

The method consists of 'unpeeling' the functions surrounding the single occurences of x by applying the inverse function to both sides of the equation. This process is called recursively until x is isolated on one side of the equation, e.g.

```
x^2 - 1 = e^3

x^2 = e^3 + 1

x = sqrt(e^3 + 1) v x = -sqrt(e^3 + 1)
```

This 'unpeeling' is done by applying a system of rewrite rules to the equation. Some examples of the rules are given in appendix II. All of them have the structure

```
P & f(U1,...Ui,...Un) = V  ->  some N in S  Ui = fi(U1,...,V,...Un)      (1)
```

where fi is the ith inverse function of f, S is a set (e.g. the integers) and N a member of that set. The existential quantifier, some, is used to indicate that we sometimes have a finite of infinite disjunction on the right hand side.

Isolation is guaranteed to succeed in solving any equation with a single occurence of the unknown, because for each R elementary function, f, the ith inverse function is also an R elementary function and there is an axiom of the form 1 in the Isolation rewrite rule set.

3.2 Collection

The purpose of Collection is not to solve equations but to reduce the number of occurences of unknowns in them to one, so that Isolation will apply. For instance, given

```
log(e,(x+1)*(x-1)) = 3
```

Collection will apply the axiom

```
(U+V)*(U-V) = U^2 - V^2                                    (2)
```

to obtain

```
log(e,x^2-1^2) = 3
```

so that Isolation can be applied.

Collection uses a system of rewrite rules like 2 above. All these axioms are of the form

```
P -> L=R
```

where some variable, U say, occurs less times in R than in L. Further examples can be found in appendix II.

Most theorem proving programs which use sets of rewrite rules apply them exhaustively, i.e. they attempt to apply each rule to every subexpression in the expression currently being manipulated. Collection uses its rewrite rule set more selectively. The rule is only applied if the variable U matches a term which actually contains the unknown x. This ensures that if Collection applies that it will succeed in reducing the number of occurences of x.

In addition the rule is only applied to a least dominating term in x,that is a term with at least two immediate subterms which contain x.

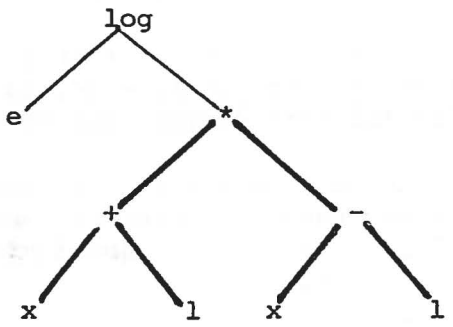

Figure 3-2: A least dominating subterm for x.

(x+1)*(x-1) is a least dominating term, whereas log(e,(x+1)*(x-1)) is not since only one immediate subterm of it contains x.

It is clear that if a Collection axiom applies to a term which is not least dominating then Isolation can be applied to that axiom to produce a Collection axiom which will apply to a least dominating subterm of the original term. All the Collection (and Attraction see below) axioms are pre-prepared so that they apply only to least dominating terms.

In section 7 we emphasize the importance of such selective applications of rewrite rules.

3.3 Attraction

Like Collection, Attraction does not solve equations, but is a service method, in this case for Collection. It brings occurences of unknowns closer together so that they can be collected. For instance, given

log(e,x+1) + log(e,x-1) = 3

Attraction will apply the axiom

log(W,U) + log(W,V) = log(W,U*V) (3)

to obtain

log(e,(x+1)*(x-1)) = 3

so that Collection can be applied.

Attraction also uses a system of rewrite rules of the form

P -> L=R

but this time both L and R must contain two variables, U and V, which are closer together in R than in L. Equation 3 above is an example and further examples can be found in appendix II.

Two subterms in an expression are closer together than two other subterms if the arc-distance between the first pair is less than the arc-distance between the second pair. The arc-distance between two terms in an expression is the number of arcs in the minimum path between the dominant functions of the terms,

in the tree representation of the expression.

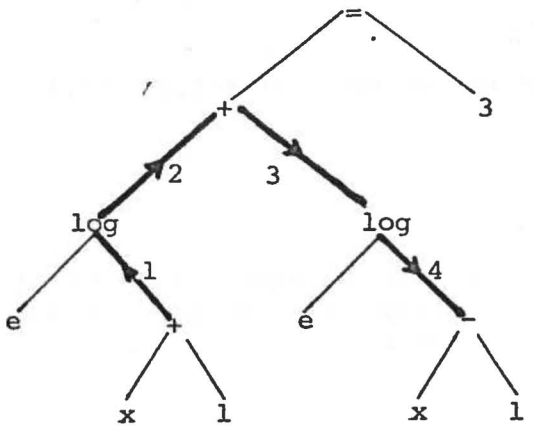

Figure 3-3: The arc-distance between the terms x+1 and x-1 is 4

As with Collection, Attraction applies its rewrite rules selectively. For a rule to be applied the two marked variables, U and V, must both match terms which contain the unknown, x. This ensures that two occurences of x are brought closer together by Attraction. Similarly, Attraction is only applied to least dominating terms in x.

3.4 Linear

Linear is a specialist method for solving Linear equations. Instead of putting linear equations in the normal form

$$a*x + b = 0$$

equations of the form L=R are solved directly. Both L and R are parsed

- to check that they are linear
- and to collect together the coefficients of x and the constant term.

These are both tidied before the answer, -b*a^-1, is returned.

Of course, Linear is redundant, since any equations it solves can also be solved by a combination of Isolation and Collection. We never-the-less included it for the sake of time efficiency.

3.5 Quadratic

Similarly, Quadratic is a specialist method for quadratic equations. Again, equations of the form L=R are parsed both to check that they are quadratic and to return the coefficients of x^2, x and the constant term. The answer is then returned in the traditional form

```
x = (-b + sqrt(b^2-4*a*c))/2*a  v  x = (-b - sqrt(b^2-4*a*c))/2*a
```

3.6 Change of Unknown

The current Change of Unknown method is very simple. It can solve equations like:

```
sin(x)^2 + 2*sin(x) + 1 = 0                                    (4)
```

by substituting y for sin(x); solving

```
y^2 + 2*y + 1 = 0
```

for y to get

```
y = 1
```

substituting sin(x) for y to get

```
sin(x) = 1
```

and solving this equation for x, to get a solution for x.

However, as in 4 the equation must have the form e(f(x)) so that substitution of y=f(x) to get e(y), is trivial. If some manipulation of the equation is required before the substitution can be made then the current version of Change of Unknown cannot proceed. This is the case, for instance, in

```
exp(5,2*x) + exp(5,x+1) + 4 = 0
```

In order to solve this equation by Change of Unknown it must first be rewritten to the form

```
exp(5,x)^2 + exp(5,x)*5 + 4 = 0
```

4. The Inequality Module

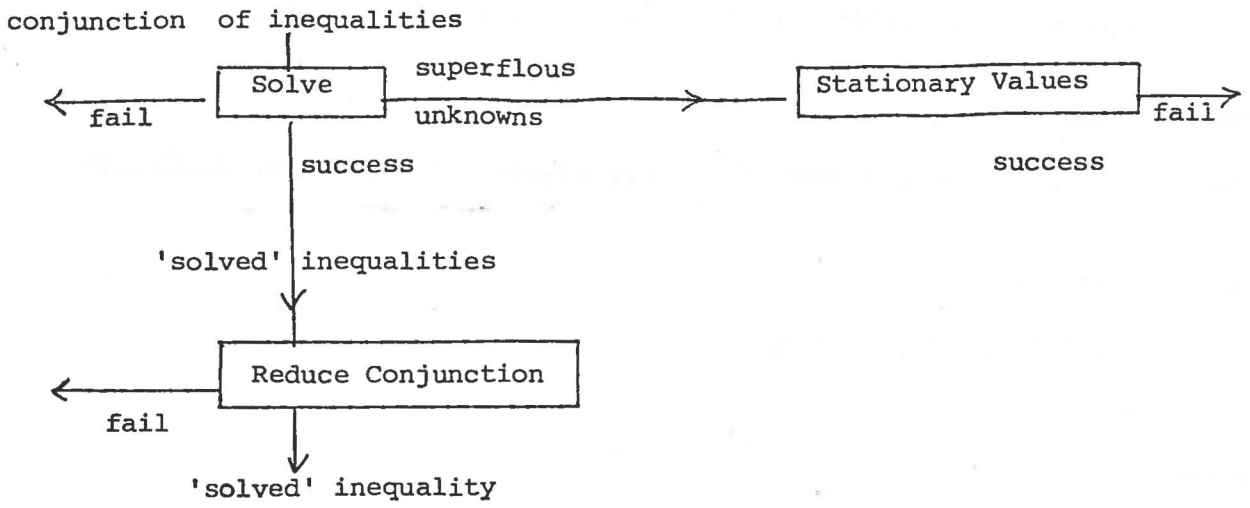

Figure 4-1: Flow Diagram: Methods of 'solving' inequalities .

Below we explain how the inequality 'solving' methods work. That is how an inequality like L>R or L>=R can be manipulated to the form x>S or x>=S, where S does not contain x. We have called this process 'solving' and S the 'solution', because of the obvious analogy between this problem and that of equation solving and the less obvious analogy between the methods which can be employed to carry out the processes. Of course, S is actually a (strict or non-strict) lower bound of x and we will sometimes also use this terminology.

Similar remarks and similar methods to those outlined below apply to the case of manipulating inequalities (L>R and L>=R) into the form S>x and S>=x.

We use four methods for 'solving' single inequalities in one unknown, one method for eliminating superflous unknowns from these 'solutions'and one method for reducing the length of conjunctions of inequalities.

The method for conjunctions only works on previously solved inequalities and is therefore applied after the other methods have been successfully applied to the individual conjuncts.

Inequalities sometimes need to be simplified and according to the circumstances the inequality module is sometimes the appropriate method for doing this. In such cases no unknown is supplied for 'solution' to be with respect to and one has to be selected by the inequality module itself. The current mechanism for doing this is very simple and consists of finding an unknown which is contained in each inequality. If non such can be found this method of simplification is abandoned.

4.1 Inequality 'Solving'

Precisely the same procedures are used to 'solve' inequalities as are used to solve equations. The pattern matching insures that only appropriate axioms are applied to the expressions at each stage and that inappropriate equation solving methods, i.e. Linear and Quadratic are not applied. The methods of: Isolation; Collection; Attraction and Change of Unknown are all appropriate and have been successfully applied to inequalities.

The method of Isolation for inequalities works exactly like the Isolation method for equations. It only applies to inequalities containing a single occurence of the unknown to be solved for and works by unpeeling functions surrounding this occurrence.

Two different, but similar, sets of rewrite rules are needed, namely those of the form

P & f(U1,...Ui,...Un) >= V -> some N in S, Ui >= fi(U1,...,V,...Un)

and

P & f(U1,...,Ui,...Un) > V -> some N in S, Ui > fi(U1,...V,...Un)

where again fi is the ith inverse function of f, S is a set and N is a member of that set.

Collection and Attraction for inequalities use exactly the same axioms as those used for equations.

The Change of Unknown method was designed to be suitable for solving both equations and inequalities. A subterm with more than one occurrence is identified and the inequality is 'solved' with respect to this. This partial solution is then solved with respect to the original unknown. e.g.

```
2*tan(x) - 1 > tan(x)
where x is acute
```

is 'solved' with respect to tan(x) to get

```
tan(x) > 1
```

then this is 'solved' with respect to x to get

```
x > 45
```

4.2 Stationary Values

Even when an inequality has been solved (e.g. by Isolation) it may not be in its desired final state. For instance, the right hand side of the inequality may contain another unknown, y , e.g.

```
x > 1/(1+sin(y)^2)                                    (5)
```

In the case of equation solving another equation would be required to eliminate y. This is not necessarily required in inequality solving. A single inequality

in two unknowns can yield a 'solution' (lower bound) for one of them which does not contain the other. , e.g. in 5 above y can be eliminated to yield

x > 1

since x is greater than 1/(1+sin(y)^2) for all values and 1/(1+sin(y)^2) attains the value 1 hence x is greater than 1.

This is achieved by

- first solving the inequality (using e.g. Isolation) to get a solution containing the second unknown.

- then applying the method of stationary values to this solution to get one independant of the second unknown

The method of stationary values consists of finding maximum, minimum and inflexion points for the 'solution', by differentiation.

- The 'solution', S, is first differentiated to form S'. S' is put equal to zero and the resulting equation, S'=0, is solved. The roots of this equation are either maximum, minimum or inflexion values.

- These values are then classified by differentiating S'(x) again to form S''(x). Each value, v, is substituted into S''(x) and v is a maximum, minimum or inflexion according as S''(v) is negative, positive or zero.

Since we are limiting ourselves to lower bounds, we are interested only in maximum values and so look only for values of v which make S''(v) negative.

A differentiation module is used to produce S' from S and S'' from S'. This is described in section 4.4 below. The equation solving module, described in section 3, is used to solve S'=0. The roots, v, are substituted for x in S''(x), using the subst procedure described in section 2.5. And the test whether S''(v) is negative is done using the simplifier (see section 5 below).

4.3 Reduce Conjunctions

Many of the MECHO problems produce a conjunction of inequalities to be solved, e.g.

```
2*g*h-2*g*r>0 & 2*g*h-4*g*r>0 & 2*g*h-2*g*r*(1+sin(a))
```

where r is non-negative

Some of these dominate others and these 'others' can be deleted, until only incomparable inequalities remain.

That this can be done is seldom immediately obvious, but it often becomes obvious when the inequalities have been 'solved', c.f. the equivalent conjunction

```
h>r & h>2*r & h>5*r/2
```

Clearly h>5*r/2 dominates h>2*r and both dominate h>r.

So after the separate conjuncts have been 'solved' an attempt is made to eliminate 'dominated' conjuncts. The basic axiom used is

```
Y>=Z  -> {(X>Y & X>Z) <-> X>Y}                                    (6)
```

but there are a variety of slight variants of 6 obtained by substituting >= for > on the right hand side and these are used too.

Using these axioms the example above can be rewritten first as

```
h>2*r & h>5*r/2
```

since 2*r>=r (r is known to be non-negative on semantic grounds (see section 5.1) and finally as

```
h>5*r/2
```

since 5*r/2 >= 2*r.

4.4 Differentiation

Writing a differentiation procedure in Prolog is simple and the result is beautiful (for examples see appendix II). The individual clauses correspond in a one to one fashion with the normal textbook rules, e.g.

```
du*v/dx = u*dv/dx +v*du/dx
```

and allowing for the Cambridge Polish notation they look very similar.

The only rule which gives any difficulty is

```
du/dv =du/dw * dw/dv
```

But even this is relatively straightforward.

- The term to be differentiated with respect to v, u(v), is checked to see if a Change of Unknown, of w=t(v), can be made, where t(v) is some proper subterm of u and all occurences of v are contained in one of the t(v).
- If this can be done then a new unknown w is created, and the substitution made, producing u'(w).
- u'(w) is then recursively differentiated with respect to w and t(v) is differentiated with respect to v.
- the two results are multiplied together.

5. The Simplification Module

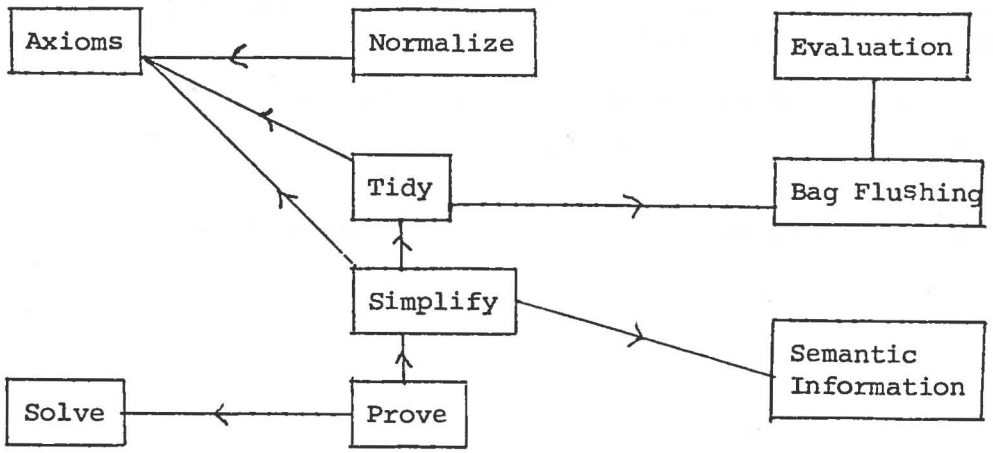

Figure 5-1: Block Diagram: Methods of simplifying expressions

The four simplification methods: Normalize; Tidy, Simplify and Prove, are used, respectively, to put expressions in a weak normal form, tidy up the output of other methods classify terms into ranges and prove simple theorems.

All the various PRESS methods expect expressions to be in a weak normal form, in which 'redundant' functions, like a-b and a/b, are replaced by the equivalent terms, a+(-b), a*b^-1 etc. To effect this replacement, Normalize uses a small set of rewrite rules which it applies exhaustively to the expression to be normalized. The rules all have the form that a redundant term appears on the left hand side with an equivalent non-redundant term on the right hand side, e.g.

U-V = U+(-V), -(U+V) = -U + -V

After the application of some of the PRESS methods, particularly Isolation, Differentiation and Substitution, expressions are often left in an untidy form, e.g.

sin(0*a + 0) ^ 1

It is the job of Tidy to clean up such expressions and present them in a simpler format. Again the main technique used is a set of rewrite rules, exhaustively applied, but arithmetic Evaluation is also used together with a special process of Bag Flushing to deal with the group theoretic operators: +, *, & and v. This time the rewrite rules define the values of algebraic functions and predicates when their arguments are zero or identity elements or

when both arguments are equal etc, e.g.

U^0=1, log(U,U)=1, U>=U <-> true

The job of Simplify is to find the range within which a term lies, e.g. to prove a term positive or an angle acute etc. To do this a set of rewrite rules is exhaustively applied and these have the effect of breaking down the problem by problem reduction. For instance, the job of proving 2*t positive is reduced to the simpler sub-problems of proving 2 and t both positive using the rule

U>0 & V>0 -> U*V>0

Simplify also uses the Tidy method together with its Bag Flushing and Evaluation sub-methods and it is able to use semantic information about the quantities in the expressions, when this is provided. These additional methods are explained more fully in the sections below.

Prove is used to prove simple theorems like 2*r>=r, which are generated by the Reduce Conjunctions and Stationary Values methods. It works by: choosing an unknown in the expression to be proved; solving the expression with respect to that unknown, using Solve, and trying to prove this result using Simplify. For instance, given the inequality

2*r>=r

the unknown r will be chosen (since it is the only candidate) and the inequality 'solved' with respect to r to yield

r>=0

This can then be proved by Simplify, using the semantic information that r is the radius of a circle and hence non-negative.

Further examples of the various simplification rewrite rules can be found in appendix II.

5.1 Using Semantic Information

The problems input to PRESS may be pure mathematical problems or they may, like the problems output by MECHO, have some meaning. This meaning may be useful for proving theorems. For instance, a symbol, t, may be the duration of a time period. If so, then it is positive and this information should be used to eliminate negative solutions for t. Similarly, a symbol, theta, may be the angle subtended by a particle travelling on the first quadrant of a circle and hence be acute. This information can be use to eliminate reflex and obtuse solutions for theta.

So that such information can be used by PRESS we have incorporated a mechanism for accessing it within the Simplify method. It consists of a special rewrite rule which given an expression returns 'true' if that expression is 'known'.

The procedure 'known' currently deals with only three kinds of expression: M>0, M>=N and N>=M, where N is a number. It accesses the MECHO database to discover information about M. All physical quantities: masses; times;

velocities etc are assumed to be positive, so that if M is a physical quantity then the expression M>0 wil be replaced by 'true'.

If the expression input is M>=N or N>=M and M is an angle then a more complex test is made.

- First M is classified to see what range it lies in, e.g. acute, obtuse etc.

- then this range information is used to try to settle the issue by showing that N lies on the appropriate side of the range, e.g. if N is 120 and M is acute then N>=M is true.

Angles are classified by examining the shape of the curve on which they are defined. This mechanism is described more fully in [Bundy et al 79].

5.2 Bag Flushing

The bag flushing module contains a collection of simplification procedures which are more easily carried out when the expression is in bag form. These procedures are: dealing with zero and unit elements; removal of cancelling pairs and evaluation. The expression is first put in bag normal form (see section 2.8) and then the procedures are applied in order.

- First the bag of arguments is searched for the occurence of a zero element. These are: 0 if the function is *; false if the function is & and true if the function is v. If a zero element is found then it is return as the value of the whole expression.

- Next any unit elements are deleted from the bag. These are: 0; 1; true and false for: +; *; & and v, respectively.

- Any arithmetic cancelling pairs: A+(-A); A*A^-1 etc are then deleted and repeated boolean pairs, A&A, AvA etc are merged.

- Finally, arithmetic evaluation is applied to all numbers in the bag. This is done not only for +, *, & and v but for the other arithmetic functions, e.g. ^, sin, log etc.

Evaluation is done, where possible, by calling the Prolog evaluable predicates (see [Pereira et al 78]), but in some cases (e.g. sin, log etc) it was necessary to write special arithmetic procedures.

6. Further Work

The work described in this report has been so fruitful in suggesting new directions that the main difficulty in writing this section has been to choose what to omit. We have decided to limit ourselves to those extensions of PRESS which are directly relevant to the use of multiple rewrite rules and meta-level descriptions - the themes of this paper. Many other possible extensions have already been described in [Bundy 75, Welham and Bundy 78, Bundy 79], and the interested reader is referred to these.

6.1 More Selective Rewriting

The normal method of applying sets of rewrite rules (exhaustive application) is to apply any rule to any term it matches. In PRESS, because each of the rewrite systems is being applied to produce a particular effect, it is often possible to be more selective about the application of rules. For instance, we have limited the application of Collection and Attraction rules to least dominating terms in the unknown x.

However, it is possible to be even more selective by restricting the number of Collection and Attraction rewrite rules we attempt to apply by using meta-level classifications of the expressions and the rules. For instance, the meta-level class of

$$(x+1)*(x-1)$$

is 'polynomial' and it is only necessary to consider applying Collection rewrite rules from the subclass 'polynomial'.

By limiting the number of possibilities in such ways, we not only cut down on the amount of search, but make feasible the use of much stronger pattern matchers, since PRESS can afford to expend more effort trying to make a rule match an expression.

6.2 Learning New Axioms

Attached to most of the sets of rewrite rules used in PRESS are inclusion and ordering criteria, simple syntactic criteria which determine when a rule is eligible for inclusion and which way round to use it (see sections 3.1, 3.2 and 3.3). These criteria are not currently represented in PRESS and the rewrite rule sets have been designed by hand.

Writing procedures to implement the criteria would be a fairly easy task. PRESS could then be made to build its own rewrite rule sets by classifying a homogenous set of axioms into subsets using the inclusion criteria and deciding which way round to use each equality and equivalence using the ordering criteria.

6.3 Suggesting Theorems to be Proved

Not only is it possible to give criteria for whether an axiom is useful to a method and in what way it is useful, it is also possible to locate gaps in the set of rewrite rules available to a method. For instance, suppose there is a function, foo, but no rewrite rule of the form

$$P \ \& \ foo(U)=V \ \rightarrow \ U=F(V) \qquad (7)$$

available to Isolation. Then it will not be possible to Isolate equations where foo is the dominating function of the left hand side. We may imagine such a situation arising naturally after the function foo has been defined. The absence of such a rule and the desire to restore the lost omnipotence of Isolation is an incentive to define an inverse for foo if one is not already defined and to try to prove a theorem of the form 7.

We hope that this process can be automated. What is needed are: monitering processes, like that outlined for Isolation above, for each of the PRESS methods; the ability to define new functions; and a theorem prover geared to proving theorems obeying a given, higly constrained, pattern.

6.4 Deriving New Methods

The Prolog clauses in appendix II which define the PRESS methods can be interpreted in two ways: as procedures for running the methods or as declarative statements in the metalanguage of algebra.

The latter interpretatation enables us to consider using them to prove properties of the methods, e.g. establish the soundness or completeness of the existing methods or to derive new methods.

We will illustrate these ideas below by showing how a simple method can be derived for equations containing two occurences of an unknown. Suppose the following facts are known to PRESS as predicate calculus clauses:

```
occ(Eqn,U,1) & isolate(U,Eqn,Ans) -> solve11(Eqn.U,Ans)

occ(Eqn2,U,2) & collect(U,Eqn2,Eqn1) -> occ(Eqn1,U,1)

collect(U,Eqn2,Eqn1) -> equiv(Eqn2,Eqn1)

equiv(Eqn2,Eqn1) & solve11(Eqn1,U,Ans) -> solve11(Eqn2,U,Ans)
```

where

- occ(exp,term,num) - means there are num occurences of term in exp.
- equiv(exp1,exp2) - means exp1 is equivalent to exp2
- isolate(unk,eqn,ans) - means ans is the result when unk is isolated in eqn.
- collect(unk,eqn2,eqn1) - means eqn1 is the result of collecting unk in eqn2.

- solvel1(eqn,unk,ans) - means ans is the result of solving eqn for unk.

These clauses can be used to derive the theorem

```
occ(Eqn2,2) & collect(U,Eqn2,Eqn1) & isolate(U,Eqn1,Ans)
                                        -> solvel1(Eqn2,U Ans)
```

and this will then become a new method for solving equations with exactly two occurences of an unknown.

7. Related Work

There are many excellent algebraic manipulation packages available, e.g. MACSYMA [Martin and Fateman 71] and REDUCE [Hearn 67]. Although PRESS has been used as the algebraic manipulation package for MECHO it was not intended primarily for such use, but as a vehicle for exploring the use of meta-level descriptions in guiding search. It is thus best seen as an extension to the use of rewrite rules for theorem proving.

Bledsoe, in various programs (see e.g. [Bledsoe and Bruell 73]), has been one of the most consistent users of rewrite rule systems and Brown (see e.g. [Brown 77]) has also been very successful. Both of these authors use single systems of rewrite rules exhaustively applied (although Brown's rules sometimes contain control instuctions within the rules themselves). These sets are very large and the criteria for including a particular rule and deciding which way round to use it (inclusion and ordering criteria) are not made explicit. This leads to several difficulties.

- Unique termination (confluence) can be proved, in theory, using the techniques of Knuth, Bendix and Huet [Huet 77], but the method is computationally expensive for large sets and is seldom done in practice.

- Proving finite termination involves defining a function on expressions and showing that it decreases when the rules are applied. This function is difficult to define when the set has no explicit inclusion criteria. On the other hand, by using their inclusion criteria, the finite termination of Isolation and Collection is trivial to establish.

- Automatic learning of new rules is not possible unless inclusion and ordering criteria are defined. For instance, Brown [Brown 77] had to order by hand all the rules 'learnt' by his system. In contrast ordering and inclusion criteria are easily defined in PRESS (see section 6.2).

- Many proofs require the application of a particular rule first one way round and later the other way. Such proofs are beyond the scope of single rewrite rule sets, exhaustively applied. However, if multiple rules are used the rule can be included in two different sets and ordered differently in each. If selective application is used, as in PRESS, then the rule can appear differently ordered in the same set, without causing looping, e.g. the axiom

  $$(U\hat{}V)\hat{}W = U\hat{}(V*W)$$

  occurs twice in the attaction set (see appendix II): left to right, to attract V and W and right to left, to attract U and V.

Boyer and Moore also used multiple rewrite rule sets in [Boyer and Moore 73], but explicit inclusion and ordering criteria were not provided and the rules were applied exhaustively.

8. Results

In this section we give a selection of the problems solved by PRESS, with comments about the processes used and timing information etc. This selection was made to try to illustrate the different abilities of PRESS, and contains some equations in one unknown, some sets of simultaneous equations in several unknowns, some inequalities and sets of inequalities.

Logarithmic Equation

```
log(e,x+1) + log(e,x-1) = 3
```

This is the equation used as a working example in [Bundy 75] and in section 3. It requires attraction followed by collection and multiple applications of isolation to produce the answer:

```
x=(e:3 + 1):2:-1 v x=(e:3 + 1):2:-1
```

This process takes PRESS 13.9 seconds of cpu time.

Exponential Equation

```
2^(x^2)^(x^3) = 2
```

This equation requires similar processes to the previous ones, but shows that these are applicable to exponential as well as logarithmic functions. The answer of

```
x=1
```

takes 5.3 seconds of cpu time to find.

Trigonometric Equation

```
( ( 2^(cos(x)^2) * 2^(sin(x)^2) )^sin(x) )^cos(x) =2^(1/4)
```

Attraction and Collection must each be applied twice to solve this trigonometric equation to produce the answer:

```
x = (180*n0 + arcsin(log(2,(2^4^-1)^2))*-1^n0) * 2^-1
where n0 is an arbitrary integer,
```

taking 76 seconds of cpu time.

Change of Unknown Equation

```
cos(x)^2 + b*cos(x) = c
```

This equation requires a Change of Unknown x to y, where y=cos(x), and the solution of a symbolic quadratic in y. The answer:

```
x = 360*n1 ± arccos(2^-1 * (-b ± (b^2+ -4*-c)^2^-1))
```

is found in 20.4 seconds of cpu time.

Simple Pulley Problem Equations

```
m1*g*cos(180) + (1*tsn + 0) = m1*(a*1) &
m2*g*1 + (cos(180)*tsn + 0) + 0 = m2*(a*1)
```

These are the equations generated by the pulley problem used as a worked example in [Bundy et al 79]. They are very easily solved by the simultaneous solution module, together with the Isolation and Linear methods to produce the answer:

```
tsn = m1*(-(-1*m1+(-m2))^-1*(g*m2+g*-1*m1))+(-m1*-1*g)  &
a = -(-*m1+(-m2))^-1*(g*m2+g*-1*m1)
```

in 15 seconds of cpu time.

Car Problem Equations

```
1760*3*d = 0*60*t + 1/2*a*60*t^2  &
v = 0 + a*60*t
```

Again these equations are generated by the MECHO program, this time from a simple constant acceleration problem. This problem requires the solution of a quadratic equation in t to produce the answers:

```
( X1 v X2 ) & ( X3  v X4 )
  where :
    X1 =  t=((-a*30)*2)^-1*((-30*a)*d*-21120)^2^-1
    X2 =  t=((-a*30)*2)^-1*(-((-30*a)*d*-21120)^2^-1)
    X3 =  v=a*(a*-30)^-1*(a*d*109312)^2^-1*30
    X4 =  v=a*(a*-30)^-1*(-(a*d*109312)^2^-1)*30
```

in 25.2 seconds of cpu time.

Stationary Values Inequality

```
x > 1/(1+sin(y)^2)
```

The method of Stationary Values has to be used to eliminate the variable y and produce the answer:

```
x > 1
```

where the value of 0 for y yields the greatest lower bound, of 1, for x. This takes 56.1 seconds of cpu time.

Great Dome Inequality

Another problem generated by the MECHO program, this time from the Great Dome Roller Coaster problem (see [Bundy et al 79]). Using the semantic information that m (as a mass) is positive d is Isolated to produce the answer:

```
d >= arcsin(3^-1 * 2)
```

in 5 seconds of cpu time.

Conjunction of Inequalities

```
2*g*h1 > 0 &
2*g*(h1-h2) >= 0 &
2*g*(h1-h2) > 0 &
2*g*(h1-h2) >= 0
```

This conjunction of simple inequalities illustates PRESS's ability to 'solve' sets of inequalities. The Isolation and Reduce Conjunction methods are used to produce the answer:

```
h1>h2
```

in 8.1 seconds of cpu time, during which h2 is assumed positive, using the semantic information that it represents a distance.

9. Conclusion

In this report we have described a computer program for doing algebraic manipulation. This program, PRESS, is based on systems of rewrite rules. Decisions about which rules are to be applied are made on the basis of meta-level reasoning about the syntactic structure of the expressions to be manipulated. The advantages of this way of organising the program are:

- The subdivision of the rewrite rules into several sets decreases the amount of search involved.

- Meta-level descriptions of the form of the rules in any set make it easy to decide which axioms to include in which set and which way round to use them. these descriptions can also often be used to establish theoretical results like termination.

- Meta-level reasoning can be used to decide which set to bring to bear at any time, and to apply selectively the rules in each set.

I. The Definition of the R Elementary Expressions

In the PRESS program we are restricting our attention to the R Elementary Expressions. A definition of this class of expressions follows:

```
<R elem exp> =:: <formula> / <terms>

<formula> =:: <formula> & <formula> / <formula> v <formula> /
              ~<formula> /
              some <var> <formula> / all <var> <formula /
              <term> = <term> / <term> >= <term> / <term> > <term>

<term> =:: <constant> / <var> / <func sym 1> (<term>) /
              <func sym 2> (<term>,<term>)

<func sym 1> =:: - / sqrt / sin / cos / tan / cosec / sec / cot /
              arcsin / arccos / arctan / arccosec / arcsec / arccot

<func sym 2> =:: + / - / * / / / ^ / log / root

<constant> =:: <real> / <arb const> / <unknown>

<real> =:: <integer> / pi / e

<integer> =:: 1 / 2 / 3 / 4 /.....

<arb const> =:: a / b / c /.......

<unknown> =:: x / y / z /.........

<var> =:: U / V / W /........
```

II. Some Samples of PRESS Code

In this appendix we give some examples of the Prolog clauses which constitute the PRESS system. We have tried to select those of particular interest and/or importance. We have removed print instructions and made some other cosmetic alterations to make them more readable.

Prolog clauses have the form

```
P :- Q, R

P :- S, T
```

where P, Q, R, S and T are atomic formulae like p(X,f(Y)). These clauses can either be read declaratively as:

```
Q & R -> P

S & T -> P
```

"Q and R imply P" and "S and T imply P"

or can be read procedurally as:

"To run P, first run Q then run R. If this fails, then run S followed by T"

Some Normalize Axioms

```
normax(U<V , V>U ).

normax(U/V , U*V^(-1) ).
```

Some Tidy Axioms

```
tidyax( U=U, true ) .

tidyax( -(-U) , U ) .
```

Some Simplify Axioms

```
tsimpax(sin(U) > 0, true) :- simplify(180>U,true), positive(U).

ntsimpax(U^(-1)>0, U>0).
```

Some Equality Isolation Rules

The first argument is the term isolated, the last is the condition.

```
isolax( U , -U=V , U= -V , true ) .

isolax( U , U+V=W , U=W+(-V) , true ) .
```

If the condition is satisfied we can avoid the negative roots
```
isolax( U , U^N=V , U=V^(N^(-1)) , (non_neg(U)& even(N))).
```

Some rules have an implicit infinite disjunction on the rhs
```
isolax( U , sin(U)=V , U=N*180+ (-1)^N*arcsin(V) , arbint(N) ) .
```

An Inequality Isolation Rules

```
isolax(U,U*V>=W, U>=W*(V^(-1)),positive(V)).
```

Some Collection Rules

The first argument is the term collected.

```
collax( W , U*W+V*W , (U+V)*W ) .
```

```
collax( U , sin(U)*cos(U) , sin(2*U)*2^ (-1) ) .
```

Some Attraction Rules

The first two arguments are the terms attracted.

```
attrax( U , V , U*W+V*W , (U+V)*W ) .
```

```
attrax( U , V , W^U*W^V , W^(U+V) ) .
```

```
attrax( U , V , log(W,U)+log(W,V) , log(W,U*V) ) .
```

```
attrax( V , W , (U^V)^W , U^(V*W) ) .
```

The same rule as above, differently ordered and differently used
```
attrax( U , V , U^(V*W) , (U^V)^W ) .
```

Some Differentiation Rules

```
diffwrtl(X^N,N*X^(N+(-1)),X) :- integer(N), !.

diffwrtl(sin(X),cos(X),X) :- !.

diffwrtl(Y*Z,Y*Z1+Z*Y1,X) :- !,
  diffwrtl(Y,Y1,X), diffwrtl(Z,Z1,X).

diffwrtl(Exp,Exp1*Arg1,X) :-
  Exp=..[Func,..Args], onexarg(Args,X,Arg),
  not atom(Arg), !, gensym(var,T),
  substl(Exp,Nexp,Arg=T), diffwrtl(Nexp,Nexp1,T),
  substl(Nexp1,Exp1,T=Arg), diffwrtl(Arg,Arg1,X).
```

The Main Simultaneous Solver Procedure

```
simsolve1(A=B & Es, [U,..Us], Ans3 & Ans2) :-
  !, solve11(A=B,U,Ans1), subst(Es,Es1,Ans1),
  gcc(simsolve1(Es1, Us, Ans2)),
  subst(Ans1,Ans3,Ans2).
```

A Key Solve Procedure

```
soll1(Eqn,U,Ans) :- collect(U,Eqn,New), soll1(New,U,Ans).
```

The Main Inequality 'Solving' Procedure

```
solveineq(Exp,X,Ans) :-
  simplify(Exp,Ineqset),
  fixvar(Ineqset,X), mapand(findbnd(X),Ineqset,Ansset),
  maximum(Ansset,Ans),
```

Some Key Substitution Procedures

```
Disjunction of substitutions
subst1(Old,New1#New2,S1#S2) :-
    !, subst1(Old,New1,S1), subst1(Old,New2,S2).

Conjunction of substitutions
subst1(Old,New,S1&S2) :-
    !, subst1(Old,Exp,S1), subst1(Exp,New,S2).

Basis case
subst1(U,Term,U=Term) :- !.

Basis case
subst1(Old,Old,U=Term) :- freeof(Old,U), !.

Recursive case
subst1(Old,New,U=Term) :- Old =.. [F,..As],
    maplist(subst1(U=Term),As,Bs), New =.. [F,..Bs].
```

Some Key Pattern Matching Procedures

```
match(E1+E2,U+V) :- var(U), var(V), !, decomp(E1+E2,[+,..As]),
    maplist(match,As,Bs), splitperm(Bs,B1,B2),
    recomp(U,[+,..B1]), recomp(V,[+,..B2]).

For matching sub-bags
changebag(Old,Left,Right,New) :-
    decomp(Old,[F,..As]), Left=..[F,..Ls],
    twofrom(As,A1,A2,Rem), Term=..[F,A1,A2],
    match(Term,Left), recomp(New,[F,Right,..Rem]).
```

Examples of the Bag Procedures

```
For putting into bag form
decomp(E+(X+Y),L) :- !, decomp(E+X+Y,L).
decomp(E+X+Y,[+,Y,..L]) :- !, decomp(E+X,[+,..L]).
decomp(E+X,[+,X,E]) :- !.

For reconstituting term
recomp(E,[+,E]) :- !.
recomp(E+X,[+,X,..L]) :- !, recomp(E,[+,..L]).
recomp(0,[+]) :- !.
```

The Main Isolation Procedure

```
isolate(U,Exp,Ans) :-
    Exp=..[Sym,Lhs,Rhs],
    singleocc(U,Lhs), freeof(U,Rhs),
    Lax=..[Sym,Left,Rhs],
    isolax(Arg,Lax,New,Condition),
    match(Lhs,Left), contains(U,Arg), Condition,
    !, tidy(New,New1),
    isolate(U,New1,Ans).
```

The Main Collection Procedure

```
collect(U,Old=0,New=0) :-
    collax(Arg,Left,Right), changebag(Old,Left,Right,New),
    contains(Arg,U), !.
```

The Main Attraction Procedure

```
attract(U,Old=0,New=0) :-
    attrax(Arg1,Arg2,Left,Right),
    changebag(Old,Left,Right,New),
    contains(Arg1,U), contains(Arg2,U),
    !.
```

The Main Stationary Values Procedure

```
findmax(Exp,X,Maxvals) :-
  diffwrt(Exp,Exp2,X),
  solve1(Exp2=0,X,Soln),
  collect_ans(X,Soln,Anslist),
  diffwrt(Exp2,Exp3,X),
  sublist(givesneg(X,Exp3),Anslist,Maxargs),
  maplist(subst2(X,Exp),Maxargs,Maxvals).
```

REFERENCES

[Bledsoe and Bruell 73]
Bledsoe, W.W. and Bruell, P.
A man machine theorem proving system
Procs of IJCAI3, Stanford, pages 56-65, 1973.

[Boyer and Moore 73]
Boyer, R.S. and Moore J.S.
Proving theorems about LISP functions
procs. of IJCAI3, Stanford, pages 486-493, August, 1973.
also available as DCL memo no. 60.

[Brown 77] Brown, F.M.
Towards the automation of Set Theory and its Logic.
Research Report No. 34, Dept. of Artificial Intelligence, May 1977.
a shortened version appeared in IJCAI5.

[Bundy 79] Bundy, A.
An elementary treatise on equation solving.
Working Paper No. 51, Dept. of Artificial Intelligence, 1979.

[Bundy 75] Bundy, A.
Analysing Mathematical Proofs (or reading between the lines)
Procs of the fourth, IJCAI, Georgia, 1975.
also available as DAI Research Report No. 2.

[Bundy et al 79]
Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M.
Mecho: A program to solve Mechanics problems.
Working Paper No. 50, Dept. of Artificial Intelligence, 1979.

[Hearn 67] Hearn, A.C.
REDUCE: A user-oriented interactive system for Algebraic simplification, pages 79-90.
In Interactive systems for experimental Applied Mathematics, Academic Press, New York , 1967.

[Huet 77] Huet, G.
Confluent reductions: Abstract properties and applications to term rewriting systems.
Rapport de Recherche 250, Laboratoire de Recherche en Informatique et Automatique, IRIA, France, August 1977.

[Martin and Fateman 71]
Martin, W.A. and Fateman, R.J.
The MACSYMA system
2nd Symposium on Symbolic Manipulation, Los Angeles, pages 59-75, 1971.

[Pereira et al 78]
Pereira, L.M., Pereira, F.C.N. and Warren, D.H.D.
User's guide to DECsystem-10 PROLOG.
, Dept. of Artificial Intelligence, Edinburgh, 1978.

[Welham and Bundy 78]
Welham, R and Bundy, A.
Equation solving: A progress report.
Working Paper No. 29, Dept. of Artificial Intelligence, June 1978.